

# Abducing Compliance of Incomplete Event Logs

Federico Chesani<sup>1</sup>, Riccardo De Masellis<sup>2</sup>, Chiara Di Francescomarino<sup>2</sup>, Chiara Ghidini<sup>2</sup>, Paola Mello<sup>1</sup>, Marco Montali<sup>3</sup>, and Sergio Tessaris<sup>3</sup>

<sup>1</sup> University of Bologna, Bologna, Italy

<sup>2</sup> FBK-IRST, Via Sommarive 18, 38050 Trento, Italy

<sup>3</sup> Free University of Bozen-Bolzano, piazza Università, 1, 39100 Bozen-Bolzano, Italy  
 {federico.chesani,paola.mello}@unibo.it, {r.demasellis,dfmchiara,ghidini}@fbk.eu,  
 {tessaris,montali}@inf.unibz.it

**Abstract.** The capability to store data about business processes execution in so-called Event Logs has brought to the diffusion of tools for the analysis of process executions and for the assessment of the *goodness* of a process model. Nonetheless, these tools are often very rigid in dealing with Event Logs that include incomplete information about the process execution. Thus, while the ability of handling incomplete event data is one of the challenges mentioned in the process mining manifesto, the evaluation of compliance of an execution trace still requires an end-to-end complete trace to be performed. This paper exploits the power of abduction to provide a flexible, yet computationally effective, framework to deal with different forms of incompleteness in an Event Log. Moreover it proposes a refinement of the classical notion of compliance into *strong* and *conditional* compliance to take into account incomplete logs. Finally, performances evaluation in an experimental setting shows the feasibility of the presented approach.

## 1 Introduction

The proliferation of IT systems able to store process executions traces in so-called event logs has originated, in the Business Process community, a quest towards tools that offer the possibility of discovering, checking the conformance and enhancing process models based on actual behaviors [1]. Focusing on conformance, that is, on a scenario where the aim is to assess how a *prescriptive* (or “de jure”) process model relates to the execution traces, a fundamental notion is the one of *trace compliance*. Compliance results can be used by business analysts to assess the goodness of a process model and understand how it relates to the actual behaviours exhibited by a company, consequently providing the basis for process re-design, governance and improvement.

The use of event logs to evaluate the goodness of a process model becomes hard and potentially misleading when the event log contains only partial information on the process execution. Thus, while the presence of non-monitorable activities (or errors in the logging procedure) makes the ability of handling incomplete event data one of the main challenges of the BP community, as mentioned in the process mining manifesto[1], still trace compliance of an execution

trace requires the presence of a complete end-to-end execution trace to be evaluated. Notable exceptions are [2,3] where trace incompleteness is managed in an algorithmic/heuristic manner using log repair techniques.

In this paper, we take an orthogonal approach and thoroughly address the problem of log incompleteness from a formal/logic-based point of view, adopting an approach based on *abduction* [4]. Differently from techniques that focus on algorithmic/heuristic repairs of an incomplete trace, we are interested in characterising the notion of incomplete log compliance by means of a sound and complete inference procedure. We rely on abduction to combine the partial knowledge about the real executions of a process as reflected by a (potentially) incomplete event log, with the background knowledge captured in a process model. In particular, abductive reasoning handles different forms of missing information by formulating *hypotheses* that explain how the event log may be “completed” with the missing information, so as to reconcile it with the process model. This leads us to refine the classical notion of *conformance-by-alignment* [5] between an execution trace and a process model into **strong** and **conditional** compliance, to account for incompleteness. In detail, the paper provides: (i) a classification of different forms of incompleteness of an event log based on three dimensions: log incompleteness, trace incompleteness, and event description incompleteness (Section 2.1); (ii) a reformulation of the notion of compliance into strong and conditional compliance (Section 2.2); (iii) an encoding of structured process models<sup>4</sup> and of event logs in the SCIFF abductive logic framework [7], and a usage of the SCIFF abductive proof procedure to compute strong, conditional and non-compliance in presence of an incomplete event log (Section 3); and (iv) an evaluation of the proposed framework in an experimental setting (Section 4). The ideas presented in the paper are illustrated by means of a simple explanatory example, and the comparison with related work is contained in Section 5.

## 2 Dealing with Incomplete Event Logs

We aim at solving the problem of identifying compliant traces in the presence of incomplete event logs, given the *prescriptive* knowledge contained in a process model. To do this, we first need to investigate what incomplete event logs are (Section 2.1) and then understand how we can adapt the notion of compliance to deal with partial data on the process execution (Section 2.2). We perform this investigation with the help of a simple example, which in this paper is described using the BPMN language<sup>5</sup>.

---

<sup>4</sup> We focus on structured process models in the spirit of [6]. Broadly speaking, this restricts to the class of models recursively composed of single-entry-single-exit blocks, where every split has a corresponding join, matching its type. This assumption rules out pathological patterns that are notoriously hard to characterise (e.g. involving nested OR joins), still providing coverage for a wide range of interesting use cases.

<sup>5</sup> For the sake of clarity we use BPMN, but our framework is language-independent.

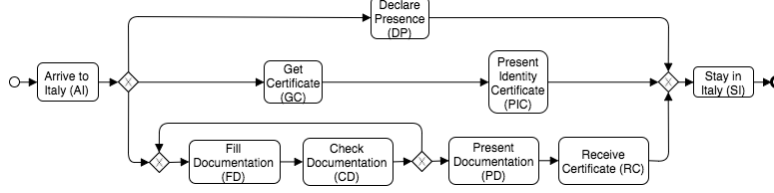


Fig. 1: A process for obtaining a permit of stay (in Italy).

*Example 1 (Obtaining a Permit of Stay in Italy).* Consider the BPMN process in Figure 1, hereafter called the Permit-Of-Stay (POS) process, which takes inspiration from the procedure for the granting of a permit of stay in Italy. Upon her arrival in Italy (AI), the person in need of a permit of stay has three different alternatives: if she is from a EU country and remains in Italy for at most 30 days, then only indicating her presence in Italy (DP) is needed; if she is from the EU and must remain in Italy for more than 30 days, then she needs to get an identity certificate (GIT) and present it (PIC). In all the remaining cases she needs to fill a documentation (FD) which is then checked (CD). When the documentation is correct, it is presented (PD) and a certificate is received (RC). The procedure concludes with the provision of the permit of stay (SI). Note that, for the sake of simplicity, the process only focuses on the so-called “happy paths”, that is, the successful issuing of a permit of stay.

## 2.1 Classifying Process Execution (In)Completeness

We assume that each execution of the POS process in Figure 1 is (partially) monitored and logged by an information system. We also assume that activities are atomic, i.e., executing an activity results in an event associated to a single timestamp: event  $(A, t)$  indicates that activity  $A$  has been executed at time  $t$ . A sample trace<sup>6</sup> that logs the execution of a POS instance is:

$$\{(AI, t_1), (FD, t_2), (CD, t_3), (PD, t_4), (RC, t_5), (SI, t_6)\} \quad (1)$$

where  $t_i > t_j$  for  $i, j \in \{1, \dots, 6\}$  such that  $i > j$ . This trace corresponds to the execution of the lower branch of the POC process, where the loop is never executed. A set of execution traces of the same process form an event log.

In many real cases, a number of difficulties may arise when exploiting the data contained in an information system in order to build an event log. For instance, data may bring only partial information in terms of which process activities have been executed and what data or artefacts they produced. Thus, instead of the extremely informative trace reported in (1), we may obtain something like:

$$\{(FD, -), (-, t_2), (SI, t_6), (-, -)\} \quad (2)$$

<sup>6</sup> We often present the events in a trace ordered according to their execution time. This is only to enhance readability since the position of an event is fully determined by its timestamp, or unknown if the timestamp is missing.

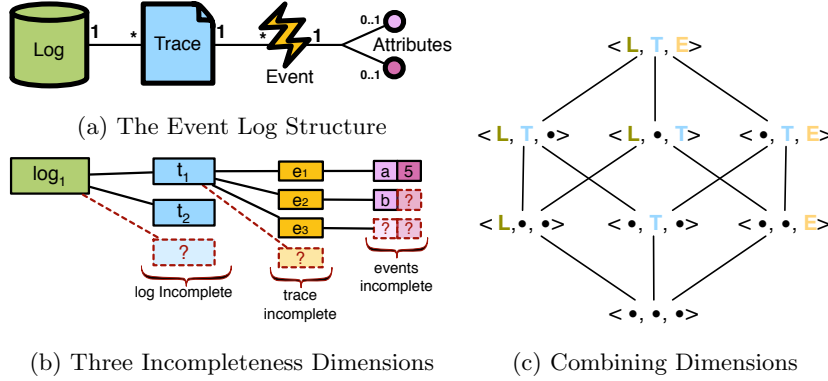


Fig. 2: Classifying (in)completeness.

This trace does not completely describe an execution of the POS process. For example, the first event logged in the trace is FD. However, by looking at the process description, it is easy to see that the first event of every execution has to be AI. By assuming that the process executors indeed followed the prescriptions of the model, this suggests that the AI-related event has not been logged. Moreover, certain events have been only partially observed. For example, the FD-related event is incomplete, because its exact timestamp is unknown. In this paper, we use “\_” to denote a missing information unit.

In accordance with the IEEE standard XES format for representing event logs [8], in general we can describe an event log as a set of execution traces. Each trace, in turn, contains events, which are described by means of n-tuples, where each element of the tuple is the value of a given attribute (see Figure 2a, where we restrict to two attributes as we do in the paper). Consequently, we can classify incompleteness along these three dimensions: incompleteness of the log, incompleteness of the trace, and incompleteness of the event description (see Figure 2b).

*(In)Completeness of the log.* Within this dimension we analyse whether all the traces envisaged by the model are in the log or not. That is, we focus on understanding whether the log contains at least one instance for each possible execution that is allowed in the model. Note that one can account for this form of (in)completeness only by: (a) limiting the analysis to the control flow, without considering complex data objects that may contain values from an unbounded domain; and (b) assuming that there is a maximum length for all traces, thus limiting the overall number of traces that may originate from the unbounded execution of loops. An example of complete log for the POS process is:

$$L_1 = \left\{ \begin{array}{l} \{(AI, t_{a1}), (DP, t_{a2}), (SI, t_{a3})\}, \\ \{(AI, t_{b1}), (GIC, t_{b2}), (PIC, t_{b3}), (SI, t_{b4})\}, \\ \{(AI, t_{c1}), (FD, t_{c2}), (CD, t_{c3}), (PD, t_{c4}), (RC, t_{c5}), (SI, t_{c6})\} \end{array} \right\} \quad (3)$$

where we assume that each trace cannot contain more than 6 event, which intuitively means that the loop is never executed twice.

Assuming this form of strict completeness is often unrealistic in practice. In fact, even under the assumption of a maximum trace length, the number of allowed traces could become extremely huge due to (bounded) loops, and the (conditional) interleavings generated by parallel blocks and or choices. Still, analysing the (in)completeness of an event log may be useful to discover parts of the control flow that never occur in practice.

*(In)completeness of the trace.* Within this dimension we focus on a single trace, examining whether it contains a sequence of events that corresponds to an execution foreseen by the process model from start to end. Trace (1) is an example of complete trace. An example of incomplete trace is:

$$\{(AI, t_1), (PIC, t_2)(SI, t_3)\} \quad (4)$$

By looking at the POS model, it is easy to see that this trace should also contain an event of the form  $(GIC, t)$ , s.t.  $t_1 < t < t_2$ .

*(In)completeness of the event description.* Within this dimension we focus on the completeness of a single event. Events are usually described as complex objects containing data about the executed activity, its time stamp, and so on [8]. These data can be missing or corrupted. As pointed out before, we consider activity names and timestamps. Thus, incompleteness in the event description may concern the activity name, its timestamp, or both. This is reflected in trace (2): *(i)* event  $(FD, -)$  indicates that activity FD has been executed, but at an unknown time; *(ii)*  $(-, t_2)$  witnesses that an activity has been executed at time  $t_2$ , but we do not know which; *(iii)*  $(-, -)$  attests that the trace contains some event, whose activity and time are unknown.

In general, we can characterise the (in)completeness of an event log in terms of (any) combination of these three basic forms. At one extreme, we may encounter a log that is complete along all three dimensions, such as the one depicted in (3). At the other extreme, we may instead have the following log:

$$L_2 = \{\{(AI, -), (-, t_{a2})\}, \{(AI, t_{b1}), (-, -), (-, t_{b2}), (SI, t_{b3})\}\} \quad (5)$$

characterised by incompleteness of the log, incompleteness of some traces, and incompleteness of some event descriptions. Intermediate situations may obviously arise as well. This is graphically depicted in the lattice of Figure 2c, where  $\langle L, T, E \rangle$  indicates the top value (completeness for all three dimensions) and  $\langle \bullet, \bullet, \bullet \rangle$  indicated the bottom value (incompleteness of all three dimensions).

## 2.2 Refining the Notion of Compliance

In our work we consider *prescriptive* process models, that is, models that describe the only acceptable executions. These corresponds to the so-called “de jure” models in [5], and consequently call for a definition of *compliance*, so as to characterise the degree to which a given trace conforms/is aligned to the

model. The traditional notion of compliance is typically considered under the assumption that the trace is a faithful footprint of reality, and requires that the trace represents an end-to-end, valid execution that can be fully replayed on the process model. We call this notion of compliance **strong** compliance. Trace (1) is an example of trace that is fully compliant to the POS process.

Strong compliance is too restrictive when the trace is possibly incomplete. In fact, the incompleteness in a trace hinders the possibility of replaying it on the process model. However, full conformance might be regained by assuming that the trace included additional activities and/or specific information on the missing data; in this case we say that the trace is **conditionally** compliant, to reflect that compliance conditionally depends on how the partial information contained in the trace is complemented with the missing one. Consider again the POS example and the partial trace:

$$\{(AI, t_1), (GIC, -)(SI, t_3)\} \quad (6)$$

It is easy to see that the observed trace is compliant with POS, *if* we assume that

$$\text{GIC was executed at a time } t_i \text{ s.t. } t_1 < t_i < t_3 \quad (7)$$

$$\text{an execution of PIC was performed at a time } t_j \text{ s.t. } t_i < t_j < t_3 \quad (8)$$

Note that the set of assumptions needed to reconstruct full conformance is not necessarily unique. This reflects that, in general, alternative strongly compliant real process executions might have led to the recorded partial trace. On the other hand, there are situations in which it is not possible to formulate additional assumptions on the partial trace to recover full conformance. In this case, the partial trace is considered **non**-compliant. For example, trace

$$\{(AI, t_1), (GIC, -)(CD, t_2)(SI, t_3)\} \quad (9)$$

does not comply with POS, since it records that GIC and CD have been both executed, although they belong to mutually exclusive branches in the model.

### 3 Abduction and Incomplete Logs

Since the aim of this paper is to provide automatic procedures, embedded in a tool, that identify compliant traces in the presence of incomplete event logs, given the *prescriptive* knowledge contained in a process model, we can schematise the input to our problem in three parts: (i) an instance-independent component, the process model, which in this paper is described using BPMN; (ii) an instance-specific component, that is, the (partial) log, and (iii) meta-information attached to the activities in the process model, indicating which are actually *always*, *never* or *possibly* observable (that is, logged) in the event log. The third component is an extension of a typical business process specification that we propose (following and extending the approach described in [9]) to provide *prescriptive* information

about the (non-) observability of activities. Thus, for instance, a business analyst will have the possibility to specify that a certain manual activity is never observable while a certain interaction with a web site is always (or possibly) observable. This information can then be used to compute the compliance of a partial trace. In fact the presence of never observable activities will trigger the need to make hypothesis on their execution (as they will never be logged in the event log), while the presence of always observable activities will trigger the need to find their corresponding event in the execution trace (to retain compliance). Note that this extension is not invasive w.r.t. current approaches to business process modelling, as we can always assume that a model where no information on observability is provided is entirely possibly observable.

Given the input of our problem, we structure this section as follows: first, we provide an overview on abduction and on how the SCIFF framework represents *always*, *never* or *possibly* observable activities; then, we show how to use SCIFF to encode a process model and a partial log (Section 3.2), third we show how we can formalize, and therefore make precise, the informal different forms of compliance presented in Section 2.2 (Section 3.3); finally, we illustrate how the SCIFF proof procedure can be used to solve the different forms of incompleteness identified in Section 2.1 (Section 3.4).

### 3.1 The SCIFF in short

Abduction is a non-monotonic reasoning process where hypotheses are made to explain observed facts [10]. While deductive reasoning focuses on deciding if a formula  $\phi$  logically follows from a set  $\Gamma$  of logical assertions known to hold, in abductive reasoning it is assumed that  $\phi$  holds (as it corresponds to a set of observed facts) but it cannot be directly inferred by  $\Gamma$ . To make  $\phi$  a consequence of  $\Gamma$ , abduction looks for a further set  $\Delta$  of hypothesis, taken from a given set of abducible  $\mathcal{A}$ , which complements  $\Gamma$  in such a way that  $\phi$  can be inferred (in symbols  $\Gamma \cup \Delta \models \phi$ ). The set  $\Delta$  is called *abductive explanation* (of  $\phi$ ). In addition,  $\Delta$  must usually satisfy a set of (domain-dependent) integrity constraints  $\mathcal{IC}$  (in symbols,  $\Gamma \cup \Delta \models \mathcal{IC}$ ). A typical integrity constraint (IC) is a *denial*, which expresses that two explanations are mutually exclusive.

Abduction has been introduced in Logic Programming in [4]. There, an *Abductive Logic Program (ALP)* is defined as a triple  $\langle \Gamma, \mathcal{A}, \mathcal{IC} \rangle$ , where: (i)  $\Gamma$  is a logic program, (ii)  $\mathcal{A}$  is a set of abducible predicates, and (iii)  $\mathcal{IC}$  a set of ICs. Given a goal  $\phi$ , abductive reasoning looks for a set of literals  $\Delta \subseteq \mathcal{A}$  such that they entail  $\phi \cup \mathcal{IC}$ .

In this paper we leverage on the SCIFF abductive logic programming framework [7], an extension of the IFF abductive proof procedure [11]. Beside the general notion of abducible, the SCIFF framework has been enriched with the notions of *happened event*, *expectation*, and *compliance* of an observed execution with a set of expectations. This makes SCIFF suitable for dealing with event log incompleteness. Let  $a$  be an event corresponding to the execution of a process ac-

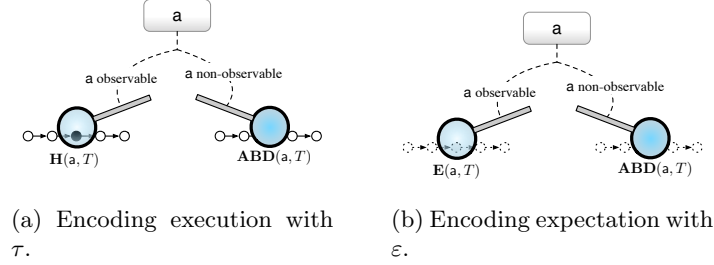


Fig. 3: Encoding always/never observable activities.

tivity, and  $T$  (possibly with subscripts) its execution time<sup>7</sup>. Abducibles are used here to make hypothesis on events that are not recorded in the examined trace. They are denoted using **ABD**( $a, T$ ). Happened events are non-abducible, and account for events that have been logged in the trace. They are denoted with **H**( $a, T$ ). Expectations **E**( $a, T$ ), instead, model events that should occur (and therefore should be present in a trace). Compliance is described in Section 3.3.

ICs in SCIFF are used to relate happened events / abduced predicates with expectations / predicates to be abduced. Specifically, an IC is a rule of the form *body*  $\rightarrow$  *head*, where *body* contains a conjunction of happened events, general abducibles, and defined predicates, while *head* contains a disjunction of conjunctions of expectations, general abducibles, and defined predicates.

### 3.2 Encoding Structured Processes and Their Executions in SCIFF

Let us illustrate how to encode all the different components of an (incomplete) event log and a structured process model one by one.

*Event Log.* A log is a set of traces, each constituted by a set of observed (atomic) events. Thus trace (4) is represented in SCIFF as  $\{\mathbf{H}(\mathbf{AI}, t_1), \mathbf{H}(\mathbf{PIC}, t_2), \mathbf{H}(\mathbf{SI}, t_3)\}$ .

*Always/never observable activities.* Coherently with the representation of an execution trace, the logging of the execution of an observable activity is represented in SCIFF using an happened event, whereas the hypothesis on the execution of a never observable activity is represented using an abducible **ABD** (see Figure 3a). Given an event **a** occurring at  $T$ , we use a function  $\tau$  that represents the execution of **a** as:

$$\tau(a, T) = \begin{cases} \mathbf{H}(a, T) & \text{if } a \text{ is observable} \\ \mathbf{ABD}(a, T) & \text{if } a \text{ is never observable} \end{cases}$$

As for expected occurrences, the encoding again depends on the observability of the activity: if the activity is observable, then its expected occurrence is mapped to a SCIFF expectation; otherwise, it is hypothesised using the aforementioned

<sup>7</sup> In the remainder of this paper we will assume that the time domain relies on natural numbers.



abducible **ABD** (see Figure 3b). To this end we use a function  $\varepsilon$  that maps the expecting of the execution of **a** at time  $T$  as follows:

$$\varepsilon(\mathbf{a}, T) = \begin{cases} \mathbf{E}(\mathbf{a}, T) & \text{if } \mathbf{a} \text{ is observable} \\ \mathbf{ABD}(\mathbf{a}, T) & \text{if } \mathbf{a} \text{ is never observable} \end{cases}$$

*Structured process model constructs.* A process model is encoded in SCIFF by generating ICs that relate the execution of an activity to the future, expected executions of further activities. In practice, each process model construct is transformed into a corresponding IC. We handle, case-by-case, all the single-entry single-exit block types of structured process models.

**Sequence.** Two activities **a** and **b** are in sequence if, whenever the first is executed, the second is expected to be executed at a later time:

$$\tau(\mathbf{a}, T_a) \rightarrow \varepsilon(\mathbf{b}, T_b) \wedge T_b > T_a. \quad (10)$$

**Xor-split** extends sequence with the possibility of selecting one among multiple target activities. In SCIFF, this is captured using an IC with a disjunction in the consequent. In particular, the fact that if **a** is executed, then either **b** or **c** is expected to be executed next is encoded as:

$$\tau(\mathbf{a}, T_a) \rightarrow \varepsilon(\mathbf{b}, T_b) \wedge T_b > T_a \vee \varepsilon(\mathbf{c}, T_c) \wedge T_c > T_a.$$

**Xor-join** indicates that, no matter which activity is executed among the input set of activities for the xor-join, then the output activity of the xor-join is expected to be executed. Hence, the encoding of xor-join can be obtained by encoding all its implied sequences. For example, if **a** or **b** are followed by **c**, we obtain:

$$\tau(\mathbf{a}, T_a) \rightarrow \varepsilon(\mathbf{c}, T_c) \wedge T_b > T_a. \quad \tau(\mathbf{b}, T_b) \rightarrow \varepsilon(\mathbf{c}, T_c) \wedge T_c > T_b.$$

**And-split** activates parallel threads spanning from the same activity. In particular, the fact that activity **a** triggers two parallel threads, one expecting the execution of **b**, and the other that of **c**, is captured using an IC with a conjunctive consequent:

$$\tau(\mathbf{a}, T_a) \rightarrow \varepsilon(\mathbf{b}, T_b) \wedge T_b > T_a \wedge \varepsilon(\mathbf{c}, T_c) \wedge T_c > T_a.$$

**And-join** mirrors the and-split, synchronizing multiple concurrent execution threads and merging them into a single thread. When activities **a** and **b** are both executed, then activity **c** is expected next, is captured using an IC with a conjunctive antecedent:

$$\tau(\mathbf{a}, T_a) \wedge \tau(\mathbf{b}, T_b) \rightarrow \varepsilon(\mathbf{c}, T_c) \wedge T_c > T_a \wedge T_c > T_b.$$

**Or-split/or-join** are captured by combining the formalization of and-/xor-elements, considering the well-known equivalence between an or-split/-join and an xor-split/-join whose alternative branches correspond to an element in the powerset of the split-targets/join-sources, whose inner activities are put in parallel. For example, the fact that **a** leads to an or-split pointing to **b** and **c** is equivalent to an xor-split that connects **a** to three outputs: one containing **b**, one containing **c**, and one containing **b** and **c** in parallel.

*Possibly observable activities.* A possibly observable activity is managed by considering the disjunctive combination of two cases: one in which it is assumed to be observable, and one in which it is assumed to be never observable. This idea is used to refine ICs used to encode the workflow constructs in the case of partial observability. For instance, if a partially observable activity appears in the antecedent of an IC, two distinct ICs are generated, one where the activity is considered to be observable (**H**), and another in which it is not (**ABD**). Thus in the case of a sequence flow from **a** to **b**, where **a** is possibly observable and **b** is observable, IC (10) generates:

$$\begin{aligned}\mathbf{H}(\mathbf{a}, T_a) &\rightarrow \varepsilon(\mathbf{b}, T_b) \wedge T_b > T_a. \\ \mathbf{ABD}(\mathbf{a}, T_a) &\rightarrow \varepsilon(\mathbf{b}, T_b) \wedge T_b > T_a.\end{aligned}$$

If multiple partially observable activities would appear in the antecedent of an IC (as, e.g., in the and-join case), then all combinations have to be considered.

Similarly, if a partially observable activity appears in the consequent of an IC, a disjunction must be inserted in the consequent, accounting for the two possibilities of observable/never observable event. If both the antecedent and consequent of an IC would contain a partially observable activity, a combination of the rules above will be used. For example, in the case of a sequence flow from **a** to **b**, where **b** is possibly observable, IC (10) generates:

$$\mathbf{H}(\mathbf{a}, T_a) \rightarrow \mathbf{E}(\mathbf{b}, T_b) \wedge T_b > T_a \vee \mathbf{ABD}(\mathbf{b}, T_b) \wedge T_b > T_a.$$

With this encoding, the SCIFF proof procedure generates firstly an abductive explanation  $\Delta$  containing an expectation about the execution of **b**. If no **b** is actually observed,  $\Delta$  is discarded, and a new abductive explanation  $\Delta'$  is generated containing the hypothesis about **b** (i.e.,  $\mathbf{ABD}(\mathbf{b}, T_b) \in \Delta'$ ). Mutual exclusion between these two possibilities is guaranteed by the SCIFF declarative semantics (cf. Definition 3).

Finally, if both the antecedent and consequent of an IC would contain a possibly observable activity, a combination of the rules above will be used.

### 3.3 Compliance in SCIFF: Declarative Semantics

We are now ready to provide a formal notion of compliance in its different forms. We do so by extending the SCIFF declarative semantics provided in [7] to incorporate log incompleteness (that is, observability features).

A structured process model corresponds to a SCIFF specification  $\mathcal{S} = \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle$ , where: (i)  $\mathcal{KB}$  is a Logic Program [12] containing the definition of accessory predicates; (ii)  $\mathcal{A} = \{\mathbf{ABD}/2, \mathbf{E}/2\}$ ; (iii)  $\mathcal{IC}$  is a set of ICs constructed by following the encoding defined in Section 3.2. A (execution) trace and an abductive explanation  $\Delta$  are defined as follows:<sup>8</sup>:

**Definition 1.** A Trace  $\mathcal{T}$  is a set of terms of type  $\mathbf{H}(e, T_i)$ , where  $e$  is a term describing the happened event, and  $T_i \in \mathbb{N}$  is the time instant at which the event occurred.

<sup>8</sup> We do not consider the abductive goal, as it is not needed for our treatment.

**Definition 2 (Abductive explanation  $\Delta$ ).** Given a SCIFF specification  $\mathcal{S}$  and a trace  $\mathcal{T}$ , a set  $\Delta \subseteq \mathcal{A}$  is an abductive explanation for  $\langle \mathcal{S}, \mathcal{T} \rangle$  if and only if

$$\text{Comp}(\mathcal{KB} \cup \mathcal{T} \cup \Delta) \cup \text{CET} \cup T_{\mathbb{N}} \models \mathcal{IC}$$

where  $\text{Comp}$  is the (two-valued) completion of a theory [13],  $\text{CET}$  stands for Clark Equational Theory [14] and  $T_{\mathbb{N}}$  is the CLP constraint theory [15] for integers.

The following definition fixes the semantics for observable events, and provides the basis for understanding the alignment of a trace with a process model.

**Definition 3 ( $\mathcal{T}$ -Fulfillment).** Given a trace  $\mathcal{T}$ , an abducible set  $\Delta$  is  $\mathcal{T}$ -fulfilled if for every event  $e \in \Delta$  and for each time  $t$ ,  $E(e, t) \in \Delta$  if and only if  $H(e, t) \in \mathcal{T}$ .

The “only if” direction defines the semantics of expectation, indicating that an expectation is fulfilled when it finds the corresponding happening event in the trace. The “if” direction captures the prescriptive nature of process models, whose *closed* nature require that only expected event may happen.

Given an abductive explanation  $\Delta$ , fulfilment acts as a *compliance classifier*, which separates the legal/correct execution traces with respect to  $\Delta$  from the wrong ones.

**Definition 4 ((Strong/Conditional) Compliance).** A trace  $\mathcal{T}$  is compliant with a SCIFF specification  $\mathcal{S}$  if there exists an abducible set  $\Delta$  such that: (i)  $\Delta$  is an abductive explanation for  $\langle \mathcal{S}, \mathcal{T} \rangle$ , and (ii)  $\Delta$  is  $\mathcal{T}$ -fulfilled. If  $\Delta$  does not contains any **ABD** then we say that it is strongly-compliant, otherwise it is conditionally-compliant.

If no abductive explanation that is also  $\mathcal{T}$ -fulfilled can be found, then  $\mathcal{T}$  is not compliant with the specification of interest. Contrariwise, the abductive explanation witnesses compliance. However, it may contain **ABD** predicates, abducted due to the incompleteness of  $\mathcal{T}$ . In fact, the presence or absence of such predicates determines whether  $\mathcal{T}$  is conditionally or strongly compliant. To make an example of how Definition 4 help us solve the problem of compliance of a single trace, let us consider traces (6), (1) and (9), of the POS example. In the case of partial trace (6), SCIFF will tell us that it is conditional compliant with the workflow model POS since  $\Delta$  will contain the formal encoding of the two abducibles (7) and (8) which provide the *abductive explanation* of trace (6). In the case of (complete) trace (1), abduction will tell us that it directly follows from  $\Gamma$  without the need of any hypothesis. The case  $\Delta = \emptyset$  coincides in fact, with the classical notion of (deductive) compliance. Finally, if we consider the partial trace (9) SCIFF will tell us that it is not possible to find any set of hypothesis  $\Delta$  that explains it. The case of no  $\Delta$  coincides, therefore with the classical notion of (deductive) non-compliance.

We close this section by briefly arguing that our approach is indeed correct. To show correctness, one may proceed in two steps: (i) prove the semantic

correctness of the encoding w.r.t. semantics of (conditional/strong) compliance; (ii) prove the correctness of the proof procedure w.r.t. the SCIFF declarative semantics. Step (i) requires to prove that a trace is (conditionally/strong) compliant (in the original execution semantics of the workflow) with a given workflow if and only if the trace is (conditionally/strong) compliant (according to the SCIFF declarative semantics) with the encoding of the workflow in SCIFF. This can be done in the spirit of [16] (where correctness is proven for declarative, constraint-based processes), by arguing that structured processes can be seen as declarative processes that only employ the “chain-response constraint” [16]. For step (ii), we rely on [7], where the soundness and completeness of SCIFF w.r.t. its declarative semantics is proved by addressing the case of closed workflow models (the trace is closed and no more events can happen anymore), as well as that of open workflow models (future events can still happen). Our declarative semantics restricts the notions of fulfilment and compliance to a specific current time  $t_c$ , i.e., to open traces: hence soundness and completeness still hold.

### 3.4 Dealing with Process Execution (In)Completeness in SCIFF

We have already illustrated, by means of the POS example, how Definition 4 can be used, at a very abstract level, to address compliance of a partial trace. In this section we illustrate more in detail how SCIFF can be used to solve the three dimensions of incompleteness identified in Section 2.1.

Trace and event incompleteness are dealt with SCIFF in a uniform manner. In fact, the **trace/event incompleteness problem** amounts to check if a given log (possibly equipped with incomplete traces/events), is compliant with a prescriptive process model. We consider as input the process model, together with information about the observability of its activities, a trace, and a maximum length for completed traces. The compliance is determined by executing the SCIFF proof procedure and evaluating possible abductive answers. We proceed as follows:

1. We automatically translate the process model with its observability meta-information into a SCIFF specification. If observability information is missing for some/all the activities, we can safely assume that some/all activities are possibly observable.
2. The SCIFF proof procedure is applied to the SCIFF specification and to the trace under observation, computing *all* the possible abductive answers  $\Delta_i$ . The maximum trace length information is used to limit the search, as in the unrestricted case the presence of loop may lead to nontermination.
3. If no abductive answer is generated, the trace is deemed as non-compliant. Otherwise, a set of abductive answers  $\{\Delta_1, \dots, \Delta_n\}$  has been found. If there exists a  $\Delta_i$  that does not contain any **ABD** predicate, then the trace is strongly compliant. The trace is conditionally compliant otherwise.

Note that, assessing strong/conditional compliance requires the computation of *all* the abductive answers, thus affecting the performances of the SCIFF proof

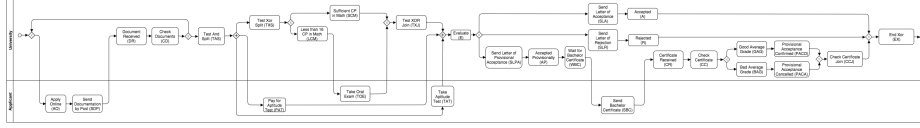


Fig. 4: University admission process

procedure. If only compliance is needed (without classifying it in strong or conditional), it is possible to compute only the first solution.

A different scenario is provided by the **log incompleteness problem**, which instead focuses on an entire event log, and looks if some possible traces allowed by the model are indeed missing in the log. In this case we consider as input the process model, a maximum length for the completed traces, and a log consisting of a number of different traces; we assume each trace is trace- and event-complete. We proceed as follows:

1. We generate the SCIFF specification from the process model, considering all activities as never observable.
2. The SCIFF proof procedure is applied to the SCIFF specification. *All* the possible abductive answers  $\Delta_i$  are computed, with maximum trace length as specified. Each answer corresponds to a different execution instance allowed by the model. Since all the activities are never observable, the generated  $\Delta_i$  will contain only **ABD**.
3. For each hypothesised trace in the set  $\{\Delta_1, \dots, \Delta_n\}$ , a corresponding, distinct trace is looked for in the log. If all the hypothesised traces have a distinct matching observed trace, then the log is deemed as complete.

Notice that, beside the completeness of the log, the proof procedure also generate the missing traces, defined as the  $\Delta_i$  that do not have a corresponding trace in the log.

## 4 Evaluation

Section 3.4 illustrates how the eight problems obtained by combining the three incompleteness dimensions can be actually solved by means of two algorithms. We now test such algorithms and study how different inputs affect their performances. As special input, we indicate whether SCIFF must compute all possible abductive explanations, or a simple yes/no answer to the compliance decision problem suffices (the latter can be answered affirmatively by stopping after having found the first abductive explanation).

For each type of incompleteness we consider, possibly only some input parameters are of interest, as, for instance, the information on the observability of activities does not impact the log incompleteness resolution (being each trace in the log assumed complete). Hence, for each problem we select the significant parameters only and perform tests by varying them in order to thoroughly understand their practical influence.

As for the model, we choose a real-life process made available within the Process Matching Contest 2013 [17], describing the admission procedure to the

Input			Output Trace Incompl.				Output Trace and Event Incompl.			
% AOA	# OE	TML	Cc	CcT (ms)	# sol.	CT (ms)	Cc	CcT (ms)	# sol.	CT (ms)
0	1	16	YES	78	2	213	YES	234	28	7069
		32	YES	109	16	915	YES	219	694	36754
	5	16	NO	551	0	551	NO	24209	0	24209
		32	YES	> 4h	6	6759	YES	> 4h	134	270278
	15	16	NO	38966	0	38966	NO	2139880	0	2139880
		32	YES	683073	36	13599417	> 4h	> 4h	> 4h	> 4h
15%	1	16	NO	266	0	20	YES	266	2	124
		32	NO	16	0	16	YES	234	2	405
	5	16	NO	548	0	548	YES	3151	1	3650
		32	YES	124	1	157	YES	2964	1	4493
	15	16	NO	43209	0	43209	YES	255483	1	28980
		32	YES	25210	36	13257351	YES	255483	1	245810
50%	1	16	NO	31	0	31	NO	156	0	156
		32	NO	31	0	31	NO	187	0	187
	5	16	NO	16	0	16	NO	202	0	202
		30	NO	31	0	31	NO	256	0	256
	15	16	YES	874	1	999	YES	9734	1	8346
		32	YES	827	1	1123	YES	12730	1	9812

Table 1: Results related to the trace and trace and event incompleteness.

Frankfurt University. Notably, in order to exercise the encoding on various process elements, a parallel branch and a loop have been added to the original procedure. Figure 4 shows the resulting model<sup>9</sup>, which is composed of 29 activities, 3 xor-splits/joins, and 1 and-split/join. If no loop iteration is considered, the model contains 8 distinct paths. The experiments have been carried out on a Windows 7 pc with 8GB RAM and a 2.4 GhZ Intel-core i7.

**Log incompleteness.** We evaluated the algorithm by varying the number of (complete) traces in the log and the bound on the length of the traces. Results shows that the number of solutions is proportional to the trace max length parameter and inversely proportional to the number of traces already presented in the log. Computing times are below 1 sec, and proportional to the number of solutions returned.

**Trace/event incompleteness.** We first feed the algorithm with a partial trace and complete events, thus testing the trace incompleteness problem. Table 1 is used to summarize the numerical values obtained, where each row represents a test case and columns report the input values used, the output and the computing time. As for the activity observability, we chose as parameter the percentage of activities that we know to be always/never observable (%AOA) and let it ranging among 0% (no certain information about activity observability is available at all), around 15% (observability is known with certainty only for a small number of activities) and 50% (for about half of the activities of the diagram we know whether they are for sure observable or not). Concerning the trace, we look at

<sup>9</sup> The model is included for providing an intuition of its complexity and no description is provided.

the number of observed events in the trace ( $\#OE$ ). Also in this case, we chose to make the parameter varying among a trace almost incomplete (1), with a small number of observed events (5) and with a medium number of observed events (15). Finally, we let the trace maximum length (TML) ranging among the values 16 (shortest path without loop) and 32 (up to 2 loops).

The table shows that the computation time when all possible completions are returned (CT) is proportional to the bound on the length of the traces (TML) and the number of completions found ( $\#sol$ ). Indeed, when one or more solutions are found, the computation time for the compliance decision problem ( $CcT$ ) is roughly  $CT/\#sol$ . On the contrary, the percentage of always observed activities in the model ( $\%AOA$ ) significantly reduces the exploration space, thus linearly decreasing the computation time. Interestingly, the more events we observe in the trace (column  $\#OE$ ) the higher the computation time is. This is because, as explained in Subsection 3.4, for each expectation about a (possibly observable) activity in the model which can potentially match (unify with) an event observed in the trace, two cases must be explored: either (i) the expectation is matched with the event in the trace, or (ii) an abductive explanation is generated for that activity. We remark that the presence of loops in the model requires both the alternatives to be explored, if we want to guarantee that *all* completions are returned.

The rightmost part of Table 1 shows the results obtained by feeding the procedure with an incomplete trace containing an incomplete event. Here, the same considerations of the trace incompleteness test described above can be drawn. The only difference lies in the fact that a high number of incomplete events (i.e., events lacking their description and, in particular, their name) let the computation time rise exponentially, as multiple possibilities for each incomplete event must be explored when looking for the set of all possible completions. Again, this is the price we pay to get the completeness of the results, which however can (in general) be avoided by asking for compliance only, as displayed by column  $CnT$  in Table 1.

We close the section by analyzing event incompleteness, that is a particular case of the trace and event incompleteness described above. In this case, the incompleteness is on the event description only (see Section 2.1), hence an input trace is characterized by a number of observed events equals to the length of the trace, and, among these, a number of missing event descriptions. From the computational viewpoint, this represents the most challenging setting, as both the above parameters cause an exponential increase in the computation times. We reckoned that for an input trace of length 16, even two missed event description brings the computation time up to a couple of hours.

#### 4.1 Discussion

The purpose of the experimentation was to stress the algorithm in borderline cases. Indeed, we remark that on the one hand we made the model convoluted on purpose especially by adding a loop, which is a source of complexity, and on the other we tested situations that are very unlikely to happen in practice. We

can safely assume that in typical scenarios the number of (partially) observable activities, which in most of the cases are human-performed, is usually no more than half of the overall activities (i.e.,  $\%AOA > 50\%$ ) and that from the  $\%AOA$  parameter and the length of the partial trace, a good estimate of the bound on the trace length can be set, thus avoiding useless loop (when present) iterations. In such settings the performance of the abductive procedure on the different types of incompleteness are reasonable. For instance, for the compliance, they range from few seconds when at most a single event description is completely unknown to about 4.5 minutes when up to 4 event descriptions are missing.

## 5 Related Work

The problem of incomplete traces has been tackled by a number of works in the field of process discovery and conformance. Some of them [2,3] have addressed the problem of aligning event logs and procedural/declarative process models [2,3]. Such works explore the search space of the set of possible moves to find the best one for aligning the log to the model. Our purpose is not managing generic misalignments between models and logs, but rather focus on a special type of incompleteness: the model is correct and the log could be incomplete.

We can divide existing works that aim at constructing possible model-compliant “worlds” out of a set of observations with incomplete information in two groups: quantitative and qualitative approaches. The former rely on the availability of a probabilistic model of execution and knowledge. For example, in [18] the authors exploit stochastic Petri nets and Bayesian Networks to recover missing information. The latter stand on the idea of describing “possible outcomes” regardless of likelihood. Among these approaches, the issue of reconstructing missing information has been tackled in [19] and in [9], respectively leveraging Satisfiability Modulo Theory, and planning techniques.

In this work, the notion of incompleteness has been investigated and extended to take into account its different variants (*log incompleteness*, *trace incompleteness* and *event incompleteness*). Similarly, the concept of *observability* has been deeper investigated and extended, by exploring the case of activities *always*, *partially* or *never* observable. This has led to a novel classification of different “degrees” of compliance.

Abduction and the SCIFF framework have been previously used to model both procedural and declarative processes. In [20], a structured workflow language has been defined, with a formal semantics in SCIFF. In [21], SCIFF has been exploited to formalize and reason about the declarative workflow language Declare.

An interesting work where trace compliance is evaluated through abduction is presented in [22]. However, they define compliance as assessing if actions were executed by users with the right permissions (auditing), and focus only on incomplete traces (with complete events), while we take a more sophisticated approach to incompleteness. The adopted abductive framework, CIFF [23], only supports ground abducibles, and ICs are limited to denials. The work in [22] explores also the dimension of human confirmation of hypotheses, and proposes



a human-based refinement cycle. This is a complementary step with our work, and would be an interesting direction for future work.

## 6 Conclusions

We have presented an abductive framework to support business process monitoring (and in particular compliance checking) by attacking the different forms of incompleteness that may be present in an event log. Concerning future development, the SCIFF framework is based on first-order logic, thus paving the way towards (i) the incorporation of data [24], (ii) extensions to further types of workflows (e.g., temporal workflows as in [25]), and (iii) towards the investigation of probabilistic models to deal with incompleteness of knowledge.

## References

1. van der Aalst, W.M.P., et al.: Process mining manifesto. In: BPM Workshops, Springer (2012)
2. Adriansyah, A., van Dongen, B.F., van der Aalst, W.M.P.: Conformance checking using cost-based fitness analysis. In: Proc. of EDOC, IEEE Computer Society (2011)
3. De Leoni, M., Maggi, F.M., van der Aalst, W.M.P.: Aligning event logs and declarative process models for conformance checking. In: Proc. of BPM, Springer (2012)
4. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive logic programming. *J. Log. Comput.* **2** (1992)
5. van der Aalst, W.M.P.: Process Mining - Discovery, Conformance and Enhancement of Business Processes. Springer (2011)
6. Kiepuszewski, B., ter Hofstede, A.H.M., Bussler, C.J.: On structured workflow modelling. In: Seminal Contributions to Information Systems Engineering. Springer (2013)
7. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: The SCIFF framework. *ACM Trans. Comput. Log.* **9** (2008)
8. on Process Mining, I.T.F.: XES Standard Definition. <http://www.xes-standard.org/> (2015)
9. Di Francescomarino, C., Ghidini, C., Tessaris, S., Sandoval, I.V.: Completing workflow traces using action languages. In: Proc. of CAiSE, Springer (2015)
10. Kakas, A.C., Mancarella, P.: Abduction and abductive logic programming. In: Proc. of ICLP. (1994)
11. Fung, T.H., Kowalski, R.A.: The iff proof procedure for abductive logic programming. *J. Log. Program.* **33** (1997)
12. Lloyd, J.W.: Foundations of Logic Programming, 2nd Edition. Springer (1987)
13. Kunen, K.: Negation in logic programming. *J. Log. Program.* **4** (1987)
14. Clark, K.L.: Negation as Failure. In: Logic and Data Bases. Plenum Press (1978)
15. Jaffar, J., Maher, M.J., Marriott, K., Stuckey, P.J.: The semantics of constraint logic programs. *J. Log. Program.* **37** (1998)
16. Montali, M.: Specification and Verification of Declarative Open Interaction Models: a Logic-Based Approach. Volume 56 of LNBIP. Springer (2010)
17. Cayoglu, U., et al.: The process model matching contest 2013 (2013)

18. Rogge-Solti, A., Mans, R.S., van der Aalst, W.M., Weske, M.: Improving documentation by repairing event logs. In: Proc. of PoEM. Springer (2013)
19. Bertoli, P., Di Francescomarino, C., Dragoni, M., Ghidini, C.: Reasoning-based techniques for dealing with incomplete business process execution traces. In: Proc. of AI\*IA, Springer (2013)
20. Chesani, F., Mello, P., Montali, M., Storari, S.: Testing careflow process execution conformance by translating a graphical language to computational logic. In: Proc. of AIME. (2007)
21. Montali, M., Pesic, M., van der Aalst, W.M.P., Chesani, F., Mello, P., Storari, S.: Declarative specification and verification of service choreographiess. *TWEB* **4** (2010)
22. Mian, U.S., den Hartog, J., S. Etalle, N.Z.: Auditing with incomplete logs. In: Proc. of HotSpot. (2015)
23. Mancarella, P., Terreni, G., Sadri, F., Toni, F., Endriss, U.: The CIFF proof procedure for abductive logic programming with constraints: Theory, implementation and experiments. *TPLP* **9** (2009)
24. De Masellis, R., Maggi, F.M., Montali, M.: Monitoring data-aware business constraints with finite state automata. In: Proc. of ICSSP, ACM Press (2014)
25. Kumar, A., Sabbella, S., Barton, R.: Managing controlled violation of temporal process constraints. In: Business Process Management. Volume 9253 of LNCS. Springer (2015) 280–296